SODAS and a methodology for system design

by DAVID L. PARNAS and JOHN A. DARRINGER
Carnegie Institute of Technology
Pittsburgh, Pennsylvania

INTRODUCTION

SODAS (Structure Oriented Description And Simulation) is a simulation language and compiler being designed at Carnegie Tech for use as a tool by the designers of computer systems. The structure of the language and its translator reflect a definition of “system” and a methodology for “system design.” It is the purpose of this paper to present the proposed methodology and language.

Definition of “system”

We shall propose a definition of “system” to be used within this paper. It is not proposed as a general definition of system, but as a local definition within this paper as an aid in discussing the design and simulation of systems. There may be systems which do not fit the definition, but it will be clear that those items usually called computer systems, either hardware or software, do fit the definition.

A system is a connected set of components whose behavior is self determined. By self determined we mean that the state and outputs of the system can be predicted from knowledge of the state of the system and the inputs. A component is a device or program which is self determined and may, in fact, be a system itself. By connected we mean that the inputs to some of the components may be the outputs of other components of the system. It should be clear that the distinction between system and component is one of level rather than substance. Any system may be a component of some other system. Ultimately, the components of the system are basic units of some sort. The only requirement that we place on these basic units is that they have well defined behavior, i.e., that it must be possible to find an algorithm which describes their behavior.

A system design methodology

A principal point in most courses in engineering analysis is the importance of having a precise definition of a problem before attacking it. Throughout such courses or textbooks this appears as a recurrent theme: A precise definition of the problem is a prerequisite to the complete understanding of the problem by the designer. A precise definition of the problem is the first step in finding the proper structuring of the problem into sub-problems. A precise definition of the problem is necessary to the evaluation of proposed solutions.

In the design of complex systems, such as computer systems, precise definitions of the goal of the design process are not easy to find. As a particularly simple example, consider the problem of designing a special purpose traffic control computer for the single intersection shown in Figure 2. Certain aspects of the problem are easy to define. The nature of the inputs to the computer can be specified, e.g., “20 inputs from sensors: 1 volt indicates the presence of a car, 0 volts the absence of a car.” The nature of the outputs (e.g., the lights to be controlled) can also be specified. It is very difficult to specify the behavior desired of the computer. One can discuss the design criteria for the behavior of the computer, e.g., it should minimize average wait time or maximize the traffic handling capacity. One can place restraints on the behavior, e.g., the maximum time for lights to remain in any mode is three minutes. Such statements do not specify the behavior; they constrain the behavior of the system. One of the design principles motivating the design of the SODAS system is that the behavior should be defined before the design of the computer itself is started.

The behavior of the system can best be described by an algorithm.* Certain variables in the algorithm can be designated as inputs or outputs, the behavior of the system described being the output response to each permissible string of input values. A designer who begins the design of a computer or similar system without such a specification has set out to solve an un-

* The algorithm need not be a computer program.
specified problem and, in all likelihood, he will design a system with suboptimal behavior (because he will confuse the question of behavior with the question of how to construct the system). The availability of an algorithm specifying the behavior makes it possible (1) to experiment (by simulation) with behavior explicitly (thereby determining exactly what is optimal behavior), and (2) to compare a proposed design with the specified behavior (to make sure that the system, as designed, accomplishes its goals).

After the behavior of the system is specified, a further structuring of the problem is desired. This usually means a specification of the various sub-systems or components and the way in which they are connected. Each of the sub-components will then become a design problem in itself. Two problems arise: the first is specifying the behavior expected of the sub-component; the second is determining that the specified sub-components, working together, will actually result in correct behavior for the whole system. Every system design team has had the experience of bringing together separately designed components only to find either (1) that the specifications for some component had been ambiguous or misunderstood or (2) that the components actually did not work together, although they did meet the specifications. The behavior of the sub-components may also be specified by algorithms. If desired, these algorithms could be verified by simulating the set of connected components and comparing the behavior of the whole system to the original specification of its behavior.

Each of the sub-components can then be broken down in the same way, until units small enough to be completely designed are obtained.

We call this approach to system design the "top down" approach. It involves starting at the "top" with a complete specification of the desired behavior of the system, then breaking the system down into smaller and smaller components until the system is specified in terms of the basic building units.

If the specifications are in languages for which computer translators are available, then it is possible to make use of simulation throughout the design process (to verify that the specifications correctly indicate the desired behavior). The specifications at every level provide evaluation criteria for the next lower level.

It is probably worthwhile noting the most obvious and important advantage of using simulation throughout all levels of the design problem. Often when the design has reached a level of great detail, decisions made at higher levels demonstrate themselves to be inconvenient and costly. If the lower levels of detail are reached only when the design has progressed to the point where a hardware implementation has begun, it is often too late to go back and make changes. If the simulation language allows the design to proceed to great detail, entirely in simulation, a new feedback loop is added to the design process. The detail design of one component can be allowed to influence a higher level design of that component or other components. The result should be a better design with fewer last minute changes.

Properties of the specification-design language and translator

In the following paragraphs we shall attempt to list and discuss the features which must be present in a specifications language and simulation system if it is to be a useful tool for "top down" system design. In a later section of the paper we shall discuss our current attempt at producing such a language and show that it provides the necessary features.

1) Designation of inputs and outputs

It is necessary to specify which variables in the algorithm describing a system are inputs and outputs. If these are not distinguished, the system is likely to be overspecified (since the designer will have to produce a component that will duplicate the behavior of the algorithm on all its variables, not simply those which will be used as inputs and outputs).

2) Combination of independently written descriptions

The language must allow the description of systems of components which have been separately described. It must be possible to take separately written algorithms, indicate the way that inputs of one are connected with the outputs of others, without excessive worry over conflicts in names of variables, etc.

3) Correct handling of simultaneous events

The translator for the language must be capable of correctly simulating simultaneous events in a system. If two of the separately described components (as mentioned above) happen to be active at the same time and interact closely, the translator must correctly simulate these simultaneous events (although it is restricted to serial execution of the individual algorithms). A method for doing this has been described in detail elsewhere.

4) Components which are themselves descriptions of systems

The language structure must be recursive; i.e., any system described in the language must be acceptable as a sub-system of a system to be described in the language. To phrase it another way, all the features available to the designer for describing the whole system must be available for describing the sub-systems.
(5) Descriptions with mixed levels of detail

The language and translator must permit the description of components of the system in varying levels of detail. The design of one component may advance faster than the design of the rest of the system. It should be possible to combine a detailed description of one component with less detailed specifications of others for testing purposes. Without this the detailed testing of components in large systems would be impossible.

(6) Mixed structural and behavioral descriptions

The language must allow structural or behavioral descriptions, where appropriate. A structural description of a system describes it as a set of components and their interconnection; a behavioral description is an algorithm which duplicates the behavior of the system. At various stages in the design process there is need for structural descriptions, behavioral descriptions and mixtures of both. The language must be designed to permit this. (The ultimate goal of the design process is usually structural description down to a low level of detail, but the description of the components at the lowest level is behavioral.)

(7) Broad class of systems

The language must allow the description of both synchronous and asynchronous discrete systems as well as analog or continuous systems and hybrid systems.

(8) Variety of languages for component description

The descriptive language and simulation system should allow a variety of means of describing the various components in the system. There are already many general purpose and special purpose simulation languages. Each of these has its own strengths and weaknesses. The language most appropriate for the description of one component may not be the same as the language most appropriate for the description of other components. It is a desirable (though not strictly necessary) feature of a simulation system that it permit the description and simulation of systems whose components are described in quite different languages. Often considerable space may be saved by simulating a part of the system in space-taking detail and the remainder at a higher level which requires less space.

Other significant design goals for the language which are not peculiar to the problem discussed here include:

(1) Small local changes in the system being described should require only small or local changes in the descriptive program.

(2) It should not be necessary to provide duplicate descriptions of duplicate components or even of components that differ only in the values of parameters.

A description of SODAS

This section is a description of the SODAS language. Several preliminary comments are in order:

(1) SODAS is an experimental language and is in the midst of its first experimental implementation. This results in several problems:

(a) Some of the sub-languages to be used in the system are not as yet defined. The grammar contains several terminal symbols indicating descriptions in other languages. These will eventually become non-terminals in the language.

(b) The language has not received actual use. Use will undoubtedly reveal restrictions and omissions in the language which will have to be removed.

We have chosen to present the language at this time in spite of its unfinished implementation because we felt that the philosophy underlying the system is sufficiently different from the underlying philosophy of other simulation languages to be worth communicating in itself. Further, we hope that interested readers not involved with the project will give us the advantage of their different viewpoints with constructive criticism.

(2) Our preliminary implementation of the language uses a slightly extended version of the Wirth-Weber precedence analyzer and parser which was described elsewhere. The class of grammars for which this analyzer is useful is only a proper subset of the class of context free grammars. In order to use the analyzer the grammar has been distorted from what might seem a more natural grammar for the language. Readers who wonder about certain parts of the grammar where the construction seems a bit strange can attribute the strangeness to our attempts to force the grammar into a restrictive mold. While it would have been possible for us to use a different grammar for expository purposes than we are using in driving our implementation, such an approach seems to introduce too many possibilities for error and has thus been avoided.

The main points contained in the grammar shown in Figure 1 are:

(1) A SODAS system is a set of sub-systems, each with specified inputs and outputs, together with a "wiring diagram" description of the way that
the components communicate.

(2) There may be only one sub-system—the system itself—or any number of sub-systems.

(3) The sub-systems may be described in any language which has been implemented in the system, including the SODAS language itself. This is a property of the simulation algorithm that is the basis of SODAS. The algorithm depends only on the existence of algorithms for simulating the sub-systems and not at all on the language in which the algorithms were originally described.

```
<SODAS descr> ::= <SFD descr> | <BOOLE descr> | <other descr> |
                   SODAS begin <dec set> <c-dec set> <components> end |
                   SODAS begin <dec set> <components> end
<dec set> ::= <dec>
<dec> ::= <wdec> | <dec> ; <wdec> | <dec> ; <i/o dec> | <i/o dec>
<type> ::= integer | real | Boolean | register
<idlist> ::= identifier | <idlist> , identifier
<iset> ::= <idlist>
<idec> ::= input <type> <idset> | input <type> array <idset> <bounds>
<odec> ::= output <type> <idset> | output <type> array <idset> <bounds>
<wdecs> ::= <wdec> | <wdecs> ; <wdec> | <wdecs> ; <i/o dec> | <i/o dec>
<reference> ::= identifier | <reference> [ <paramlist> ]
<refer> ::= <reference>
<dep dec> ::= <refer> ← . <refer>
<dep dec> ::= <dep decset>
<dep decset> ::= <dep dec> ; <dep decset> ; <dep dec>
<b-pairset> ::= <constant>; <constant> | <b-pairset> , <constant>; <constant>
<b-pairs> ::= <b-pairset>
<bound> ::= [ <b-pairs> ]
<comp dec> ::= subsystem <c-ident> ( <idset> ) <i/o dec set> <dep decs> <comp body> |
                   subsystem <c-ident> <i/o dec set> <dep decs> <comp body> |
                   subsystem <c-ident> <i/o dec set> <comp body> |
                   subsystem <c-ident> ( <idset> ) <i/o dec set> <comp body>
<comp body> ::= <SODAS descr> | clock<constant><SODAS descr>
<component> ::= subsystem <i/o dec set> <connect set> <comp body> |
                   subsystem <i/o dec set> <connect set> <dep dec> <comp body> |
                   subsystem <c-ident> ( <paramlist> ) <connect set> |
                   subsystem <c-ident> <connect set>
<c-ident> ::= identifier
<comp set> ::= <component> | <comp set> ; <component>
<components> ::= <comp set>
<con grp> ::= <refer> ← <refer> | <refer> → <refer> |
               <con grp> ∧ <refer> → <refer> | <con grp> ∧ <refer> ← <refer>
<connect set> ::= <con grp>
<params> ::= <identifier> | <constant> | <params> , <constant> |
               <identifier> | <params>
<paramlist> ::= <params>
<c-dec set> ::= <c-decs>
<c-decs> ::= <comp dec> ; <c-decs> | <comp dec>
<i/o dec> ::= <idec> | <odec>
<i/o decs> ::= <i/o dec> ; <i/o dec>
<i/o decs> ::= <i/o dec> ; <i/o dec> ; <i/o dec>
```

Figure 1—Grammar for SODAS
There is, however, much information about the language which cannot be contained in the syntax.

The language is designed for use in simulating discrete systems. These are systems which can be considered as changing state only at discrete points in time known as "clock pulses." As is always the case in simulation on digital computers, systems whose state variables are changing continuously over periods of time can be simulated only to the extent that they can be approximated as discrete systems.*

The algorithms describing the behavior of a system or component describe the behavior on each clock pulse. The variables in the algorithm are the memory of the component, the information that it carries from clock pulse to clock pulse. Inputs are read and outputs computed on each clock pulse. It is, however, possible for an algorithm to indicate that it will be inactive for a fixed number of clock pulses or until certain external conditions hold. The simulation system can take account of this in determining an efficient way to simulate the system.

To correctly simulate simultaneous events by simulating the individual events in sequence, it is sometimes necessary to simulate one component, restore it to its previous state, and later simulate it again.

Therefore, the simulation algorithm requires of a sub-language that its compiled algorithm can be simulated without a permanent change in the state variables or memory of the algorithm. Since this is the only firm requirement of sub-languages, it is possible for sub-language compilers to be written independently of the SODAS compiler itself.

We shall now consider the interpretation of the various syntactic units of the language.

SODAS DESCR: A SODAS description may be a description in any of the various other languages included in the system, or it may have the form:

```
SODAS BEGIN <DEC-SET> <C-DEC-SET> <COMPONENTS> END
```

The terms in italic are terminal symbols in the grammar and have no syntactic definition in the grammar.

DEC SET: A DEC SET is a set of declarations much like ALGOL declarations. Variables may be declared as inputs, outputs, or simply interconnecting variables. Those declared to be inputs and outputs are for use in communicating with external systems; the others are used later in describing the way that the components are connected. They may be thought of as patch cords which are used to connect the output of one component to the input of another.

C-DEC SET: A C-DEC SET is the set of component declarations. Any component or type of component which will be used more than once in the system can be declared here and given a name and optional parameters. The name can be used later as a substitute for a description of the component. If the parameters are given, then the declaration is a description of a class of components. A member of this class is determined by supplying values for the parameters during the use of the component. Separate uses of the component or members of the class of components are completely independent. They have no common memory or interconnectors. They are simply copies of a common template.

The form of a component declaration is:

```
SUBSYSTEM <C-IDENT> ( <IDSET>) <I/O DEC-SET> <DEP DECS> <COMP BODY>
```

C-IDENT: The C-IDENT is the identifier assigned to the component or class of components. The IDSET is simply the (possibly empty) list of parameters for a declaration which defines a class of components. The I/O DEC SET is a set of declarations (as described above) which indicate the inputs and outputs of the component.

DEP DECS: The set of statements known as DEP DECS is a description of some basic properties of the component. It is possible in SODAS to deal with devices that have instantaneous response. An output of a device at some discrete point in time can be a function of the inputs to that device at the same time. SODAS is capable of simulating such components and systems correctly, but it must have information about such immediate dependencies.

A DEP DEC is a statement of the form:

```
(Output) (Input)
IDENTIFIER • IDENTIFIER
```

The variable on the left must be an output of the component, the variable on the right an input to the component. Such a declaration indicates that the value of the component on the left may be a delayless function of the input indicated. If the statement is not true it may result in a less efficient simulation than necessary, or in some cases in the SODAS system's deciding that the system cannot be simulated. An unnecessary statement will never produce an incorrect simulation, but a missing statement might do so. It is planned to allow such declarations to be conditional in future implementations—this will expand the class of systems that SODAS can deal with. At present the syntax only allows unconditional declarations.

The component body is a system description, either in SODAS or in some other system.

---

*This restriction is fundamental to the problem of simulating continuous change on a digital computer. It is not a special restriction in SODAS, nor is it a disabling restriction. We are as close to achieving design goal (7) as is possible.
COMPONENTS: The syntactic type COMPONENTS is the set of all the parts of the sub-system. Each component is either an instance of a declared component type or is described in the body of the system description. Components which are instances of a type already declared are specified by the identifier used in their declaration and the values of the parameters, if any. The declaration and body of other components appear in the system description. As each component is described, its connections are also described. A connection is described by statements of the form

\[
\text{(Input)} \quad \text{IDENTIFIER} \leftarrow \text{IDENTIFIER} \\
\text{(Interconnector)}
\]

where the identifier on the left is an input variable for the component, and the identifier on the right is an interconnector (or wire) in the system, or by a statement of the form

\[
\text{(Output)} \quad \text{IDENTIFIER} \rightarrow \text{IDENTIFIER} \\
\text{(Interconnector)}
\]

where the identifier on the left is an output of the component and the one on the right is an interconnector.

By consistently placing the interconnector on the right side of the statement, it is possible to resolve any possible conflicts between the names of interconnectors in the system and inputs to the component.

In the following we find it convenient to use the term "parent" system to describe the smallest system which contains a given system as a component. Any system which contains the given system as a component may be termed an ancestor.

Timing

It was mentioned earlier that each component of a system is described by an algorithm which performs the actions taken on a single clock pulse each time it is executed. The most straightforward method of simulating such systems would be to execute every algorithm on every simulated clock pulse. Such a simulation would often be a very inefficient simulation of a system. SODAS has several descriptive features which avoid such simulations where possible.

The systems simulated in SODAS are discrete systems in that activity is presumed to occur only at discrete points in time. The system as a whole can be thought of as being driven by an external clock pulse, but the clock pulses need not be thought of as equally spaced in time. There must be a clock pulse whenever activity occurs, but these periods of activity need not be at integer multiples of some basic time unit. This facilitates simulation of systems whose sub-systems are not naturally described on the same time scale.

Systems in SODAS may be either synchronous or non-synchronous. In a synchronous system all sub-systems or components are assumed to be clocked by a master clock of the system. Activity takes place only on clock pulses which occur once every basic time unit. For each sub-system, the time between clock pulses is either the same as that of the parent system or some integer multiple of it.

In non-synchronous systems each component determines its own periods of activity and inactivity. If all components of a system are inactive, the system is inactive.

Both types of system may be simulated in SODAS. A SODAS system is synchronous if all of its sub-systems are synchronous. A non-synchronous system may have synchronous sub-systems, but a synchronous system may not have non-synchronous sub-systems.

Both synchronous and non-synchronous systems may be described as having a clock interval which is a constant multiple of the clock interval of the parent system. For synchronous systems the clock interval of a component must be an integer multiple of the clock interval of the system. For non-synchronous systems the interpretation of the clock rate description is quite different. Each non-synchronous component must keep track of the simulated time. The clock rate description simply indicates a difference in scale between the time measures used within the component and that used in the parent system.

Both synchronous and non-synchronous systems may have components which indicate that they are to be inactive for periods of time. For a synchronous system the inactive periods must be integer multiples of the basic time unit, but this restriction is not needed for non-synchronous systems.

Both synchronous and non-synchronous sub-systems may indicate that they will be inactive until some Boolean expression involving inputs holds. This is equivalent to the wait until of SOL. A system is active only if at least one of its components is active. Thus, if all components of a system are inactive, the system need not be simulated. If one or more components are doing a "wait until," and their Boolean expressions include inputs external to the parent, then if all sub-systems are inactive, the parent system must report to the ancestor system that it is doing a "wait until," and indicate the variables involved.

It is hoped that the above provides a sufficient set of mechanisms to allow efficient simulation of sub-systems which are essentially inactive (either marking time or waiting for an external event). With this feature, one can approach the efficiency of systems like GPSS and SIMSCRIPT. The unique feature of the simulation method used in SODAS is its behavior when several components are active at once. At this point SODAS can determine a correct order for simulating the sub-systems.
We are including an example to demonstrate the use of SODAS. Since the SODAS system is not yet operational, we have had to simulate its behavior using Carnegie Tech's ALGOL and BOOLE^4 translators. The problem is not claimed to be a practical problem, nor is it touted as an example of a particularly good design. It was chosen because it was small enough for presentation and assimilation in a relatively short period of time, yet large enough to illustrate the features of SODAS. The example is presented in considerable detail so that those who wish to study the way that the language is used may do so.

The problem that will be discussed was first presented in a Carnegie Institute of Technology course on logic design in 1963. In this course, the students must completely design a working system in terms of idealized logic. The problem constitutes a term project for groups of three students who were seniors in Electrical Engineering. In 1963, the majority of the groups completed the design in the relatively short period of time allotted. SODAS was not available to the students. This project, and those of later years, gave the senior author the opportunity of observing a large number of design teams working on the same design project. This observation led to a number of conclusions:

1. Groups which used the "bottom up" approach, designing small parts of the project and trying to put them together into a total system, did not produce working systems.

2. Among the groups that were successful, the work was generally quite poor, unstructured, and incorrect, unless they either hit upon (or were led to) the top-down approach. All work before that could be classified as false starts.

3. After the top-down approach was started, the work progressed well until it became time to test the unit composed of the components designed separately by the various members of the group. There were two distinct sorts of difficulties:

   a. Components which should have worked together did not. The students did not have a precise way of communicating the specifications of the components to each other. When they came down to the moment of truth, it was apparent that each had his own idea of the division of labor. As a result, extensive last minute redesign was required. A few groups did not complete the project due to this factor.

   b. Even though the individual components met their specifications, the combined system failed. The difficulty was that the students had no means of verifying that their initial structuring of the problem was correct and feasible. Some oversight at the earlier "behavior specification of sub-components" stage had not been detectable until the components were designed and being tested. These errors were often very difficult to recover from, and the redesigned systems that did work often were obtained at the price of poor performance by the system.

We have no doubt that the SODAS system would have been a great deal of aid to these students—enough aid to turn what was, for them, a very difficult problem into a relatively easy problem. The authors feel quite strongly that such problems are not confined to students in courses, but are common in professional circles as well.

The problem described below is deceptively simple. Assigned to the class mentioned, without SODAS, it was a problem that was probably too large for the time allowed to the students. With the aid of SODAS and the methodology discussed above, it now appears far too simple for the course. This may only be a matter of appearance, resulting from our increased familiarity with the problem; only actual experiments in design with the aid of SODAS will test our conjecture that the increased simplicity of the problem results from the availability of the new system.

An example: design of a traffic control module

A modular design of a traffic control system would include traffic control modules at each intersection and a master control unit to supervise the overall operation. In the following example we are going to look at possible specifications for an individual module and follow through the design of some of the actual hardware—using the "SODAS Method" of system design.

Specifications for traffic control module

Each module is to control a single intersection such as is shown in Figure 2 and be easily adaptable to a simpler intersection. It is to operate with various degrees of influence from the master unit, ranging from complete outside control to independent operation. In addition to controlling the traffic lights, the module is to supply information about the local traffic conditions to the master unit.

To limit the complexity of this example, the module will use only the two sets of modes (traffic flow patterns) shown in Figure 3. The first set (modes 1 through 4) is to be used if the intersection does not have a separate left turn lane. Otherwise, the second set (modes 5 through 8) is to be used.
Figure 2—Model intersection

Control module inputs (from sensors and master control unit)

SENSOR [1:20] — Sensor inputs (See Figure 2.)

OUTSIDE LIGHT [1:12] — Master unit's control lines for the lights

TMAX — Maximum time in any mode (traffic flow pattern)

TMIN — Minimum time in any mode

CYCLETIME — Total time to cycle through all 4 modes
SODAS and a Methodology for System Design

Figure 3—Traffic flow patterns

INITIALMODE — Initial mode
COC — Complete outside control switch
LOC — Limited outside control switch
RESET — Initialization switch

Control module outputs (to lights and master control unit)

CARS [1:4] — Number of cars waiting in each leg of intersection
LEFTTURN [1:4] — Lines indicating whether cars are waiting in left-turn lanes
LIGHT [1:12] — Signals controlling the 12 traffic lights.

Through inputs COC and LOC the master unit indicates one of three possible degrees of outside control: complete outside control, limited outside control, independent operation. When the module is operating independently, it is expected to keep account of the number of cars waiting to pass through the intersection and to control the lights appropriately. The control is, however, to be subject to two parameters, the minimum time and maximum time to spend in any one mode (condition of the traffic lights). This will assure the driver of a minimum amount of time to pass through the intersection, once he receives a “green light,” and also assure him of a maximum waiting time (when he is on a little-used road intersecting one which is very heavily used).

The LOC input signal indicates that the module is to leave its completely independent operation and submit to limited outside control. In this type of operation, the module is still partially in control, but is subject to a basic cycle time imposed by the master unit in order to synchronize it with other units. The basic cycle time is the total time in which the unit must complete its cycle of all four modes shown in Figure 3. At this time, of course, the master unit can also set the maximum and minimum times to values which will give the module the degree of freedom desired. At the time of initiating outside control the master control unit indicates the initial mode for the module, thus bringing about complete synchronization. Provision is also made for direct control of the lights by the master unit (OUTSIDE [1:12]). For each of the 12 lights at the intersection, a line is provided to signal TRUE for green and FALSE for red. It is assumed that the traf-
Traffic lights have mechanical timers that control the yellow light during the change.

The clock frequently is to be 10 Hz. Although usually slow, it assures that all cars will be detected correctly at speeds less than 100 mph.

**First level of design**

The first step taken was to treat the module as the black box shown in Figure 4 and to write a SODAS program describing its inputs, outputs, and behavior. This initial description was written in SFD-ALGOL, an ALGOL-like sub-language of SODAS with facilities for specifying inputs, outputs, and timing.

In conjunction with the above, a traffic simulator was written. Such a simulator is most easily described in a SODAS sub-language resembling SIMULA and is treated as a component in the total system.

The entire system, including the traffic, was then simulated and changes were made in the algorithm that selects the traffic flow pattern until reasonable traffic flow was obtained. Since, at this stage, the control algorithm was described in an ALGOL-like language and not embedded in hardware, such testing and changing was a simple task. Figure 5 contains the SODAS description of the traffic control system at this point.

**Second level of design**

At the second level of design, structural information was added to the behavioral description obtained above.

The module is required to count the cars waiting in each leg of the intersection at all times. Since this task is identical for each leg and since such a counter is easily isolated from the rest of the system, it was decided to design four counters as sub-systems.

The single sensor in the left-turn lane makes the counting of cars more involved. It is assumed that if a car were waiting in the left-turn lane and there were a green light on the last clock pulse, it would leave on this clock pulse. This assumption forces the counter for each leg to record the "left-turn light" to account for cars leaving the leg through the left turn lane.

The module initially selects its traffic flow pattern from the first set (Figure 3). It assumes that no left-turn lane exists until one of the left-turn sensors is activated. These signals are combined by a "joiner" (actually an "or" gate). Figure 6 indicates the structure introduced at this point.

In the first level of design, the task of counting cars was handled by the procedure "count." However, as written, this procedure had to be executed twice for each simulated clock pulse—once to update the car-count and again to store the value of the left-turn light after it was determined.

At the second level of design, this problem is reflected in the fact that the values for LIGHT [1:12] (counter input) are declared immediately dependent on CARS [1:4] (counter output). This immediate dependency and others are indicated in the block diagram (Figure 6) by dotted lines and in the SODAS description (Figure 7) by dependency statements.

The reader should note the close correspondence between the block diagram and the SODAS description. Since the four counters are identical, only one needs to be described (by a component declaration). Then the four counter descriptions are references to that declaration.

**Third level of design**

The third step in the design is the final step in our example. In it the counter for leg 1 is divided into four sub-systems and each of these is implemented in idealized logic elements (flip-flops and gates). Although we could have introduced one more level by describing the components of the counter in SFD-ALGOL, these descriptions would have been extremely simple and it seemed more reasonable to implement them directly using logic elements.

Of the five sensor inputs to the counter for leg 1, two (1, 2) indicate cars entering the leg and two (3, 4)
### SODAS Description

**SFD** begin  
input Boolean array sensor [1:20];  
input Boolean array outsideight [1:12];  
input Boolean loc, coc, reset;  
input integer tmax, tmin, ttot, initialmode;  
output Boolean array light [1:12];  
output Boolean array leftturn [1:4];  
output integer array cars [1:4];  
integer mode, tim, ctmax, ctmin, cttot;  
Boolean ltl;  
real factor;

**procedure** initializecontroller;  
begin  
integer i;  
for i~1,2,3,4 do  
begin  
  cars[i]=0;  
  leftturn [i]=false;  
end;  
for i~1 step 1 until 12 do light[i]=false;  
mode ← initialmode;  
tim←tmax; factor←1;  
ctmax←tmax; ctmin←min;  
cttot←if loc then ttot else 1800;  
ltl←false;  
end initialization of the controller;

**procedure** count (a); Boolean a;  
begin  
  own Boolean array lastleftturn [1:4];  
  integer i;  
  for i~1,2,3,4 do  
  if a then  
  begin  
    for j~4,3 do if sensor [5*i-j] then  
    cars [i] ← cars [i] + 1;  
    for j~2,1 do if sensor [5*i-j] then  
    cars [i] ← cars [i] - 1;  
    if sensor [5*i] then leftturn [i] ← ltl ← true;  
    if leftturn [i] ∧ lastleftlight [i] then  
    begin  
      cars [i] ← cars [i]-1;  
      leftturn [i] ← false;  
    end;  
    end else lastleftlight [i] ← light [3*i];  
  end count;

**Comments**

Declaration of input, output and internal variable.

**procedure** initializecontroller; initializes internal variables (memory) and output variables.

Procedure count (a); Boolean a; if the parameter a is true, the number of cars waiting on each leg is updated and a left-turn lane is checked for. If a is false, the values of the left-turn lights are recorded to be used on the next clock pulse.

**Figure 5**—SODAS description of traffic control module at level one (Pts I, II, III)
SODAS Description

procedure setlights (a,b); Boolean array a;
integer b;
begin
  integer i;
  for i=1 step 1 until 12 do a[i]~false;
end setlights;

integer procedure newmode;
begin
  integer i, high, nscars, ewcars, tcars;
  nscars=cars[1] + cars[3];
  ewcars=cars[2] + cars[4];
  tcars=ewcars + nscars;
  if tcars=0 then tcars=1;
  if ¬ltl then
  begin
    high=0;
    for i=1,2,3,4 do
      if i = mode ∧ (cars[i]>0 ∨ loc) then
        begin
          if cars[i]>high then
            begin
              high=cars[i]; newmode=i;
            end;
          end;
          factor=high/tcars;
        end else
        begin
          if mode = 5 ∨ mode = 7 then newmode=7; factor=5*nscars/tcars; end else
          if mode = 5 ∨ mode = 7 ∧ (leftturn[2]∨leftturn[4]) then
            begin
              newmode=5; factor=5*ewcars/tcars; end else
          if mode = 8 ∧ (nscars>ewcars) then
            begin
              newmode=8; factor=nscars/tcars; end else
          begin
            newmode=6; factor=ewcars/tcars;
          end;
        end of new mode procedure;
      end;
end newmode;

integer procedure cycletime;
if mode = 5 ∨ mode = 7 then cycletime ← tmin
else if ¬loc then cycletime ← tmax
else cycletime ← factor*cttot;

Comments

sets the traffic lights given a desired traffic flow pattern (mode)
selects new mode to maximize the number of cars allowed to move
determines the time to be spent in the new mode
SODAS Description

continue: time begin
if reset then initializecontroller;
if coc then
begin integer i;
for i=1 step 1 until 12 do
light [i] ← outsidelight [i];
go to done;
end;

if loc then cttot ← ttot else cttot ← 1800;
tim ← tim + 1;
count (true);
if tim > ctmax then
begin
tim ← 1;
mode ← newmode;
ctmax ← cycletime;
setlights (light, mode);
count (false);
end;
done: go to continue;
end time block;
end SFD ALGOL description of traffic control module

— start of clock pulse
under complete outside control lights as set by master control's input
under limited outside control the master unit sets the total cycttime
tim is time in mode update car count
after specified time select new mode, new cycle-time, and store new light values for counter
— end of clock pulse

indicate cars leaving. SENSOR [5] also indicates cars leaving but only if the left-turn light is green. Thus at a single clock pulse it is possible for the net change in the number of cars waiting to be +2, +1, 0, −1, −2, −3.

In looking for a structure for the counter, the following was noted:

1. A binary to decimal conversion would be needed to allow communication between the binary outputs of the counter and the decimal inputs of the rest of the system.
2. A special counter would be needed that could accept increments of 1 or 2 and decrements of 1, 2, or 3 in a single clock pulse.
3. A network would be needed to decide
   (i) if a left-turn lane existed
   (ii) if cars were waiting in the left-turn lane
   (iii) if a car had left via the left-turn lane
4. A decoding network would be needed to interpret the signals from the sensors and compute the net change in number of cars waiting.

Figure 8 shows the block diagram of the counter for leg 1 incorporating the structure discussed above.
The “special counter” is a 7-bit counter that ignores inputs attempting to take its value below zero or over 127. This was done to limit error due to incorrect inputs or inadequate counter capacity.
The binary to decimal conversion was described in SFD-ALGOL while the special counter, decoder, memory unit, and the joiner (Figure 6) were designed using standard logic design techniques and described in BOOLE,4 another sub-language of SODAS.

Figures 9 through 12 show the structural diagrams of the three counter components implemented in logic elements. Studying these figures and the SODAS description of the memory unit at this level in Figure 13,
Figure 6—Second level of description
### SODAS Description

<table>
<thead>
<tr>
<th>SODAS begin</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>input</strong> Boolean array sensor [1:20];</td>
<td>declaration of inputs, outputs, and interconnectors for the traffic control module</td>
</tr>
<tr>
<td><strong>input</strong> Boolean array outsidelight [1:12];</td>
<td></td>
</tr>
<tr>
<td><strong>input</strong> Boolean loc, coc, reset;</td>
<td></td>
</tr>
<tr>
<td><strong>input</strong> integer tmax, tmin, cycletime, initialmode;</td>
<td></td>
</tr>
<tr>
<td><strong>output</strong> Boolean array light [1:12];</td>
<td></td>
</tr>
<tr>
<td><strong>output</strong> Boolean array leftturn [1:4];</td>
<td></td>
</tr>
<tr>
<td><strong>output</strong> integer array cars [1:4];</td>
<td></td>
</tr>
<tr>
<td><strong>output</strong> Boolean array L[1:4]</td>
<td></td>
</tr>
</tbody>
</table>

**subsystem counter**

| input Boolean array sensor [1:5]; | |
| input Boolean reset, light; | declaration of subsystem inputs and outputs |
| output Boolean ltl, leftturn; | declaration of immediate dependencies |
| output integer cars | |
| lt1 ← sensor; lt1 ← reset; | |
| leftturn ← sensor; leftturn ← reset; | |
| cars ← sensor; cars ← reset | |

**SFD begin**

| input Boolean array sensor [1:5]; | |
| input Boolean reset, light; | |
| output Boolean ltl, leftturn; | Note #1 |
| output integer cars; | |
| integer i, j; | |
| Boolean lastleftlight; | |
| **1a: time** begin | |
| if reset then | updates count of cars waiting in each leg, indicates if left-turn lane exists and stores current values of left-turn lights |
| begin | |
| leftturn ← false; | |
| ltl ← false; | |
| cars ← 0; | |
| end; | |
| for i=1,2 do if sensor [i] then | | |
| cars ← cars + 1; | | |
| for i=3,4 do if sensor [i] then | | |
| cars ← cars - 1; | | |
| if sensor [5] then leftturn ← ltl ← true; | | |
| if leftturn ∧ lastleftlight then | | |
| begin | | |
| leftturn ← false; | | |
| cars ← cars - 1; | | |
| end; | | |
| lastleftlight ← light; go to 1a; | | |
| end of time block; | | |
| end SFD ALCOL description | | |

**Note #1:** This redeclaration of inputs and outputs allows the sublanguage compiler to operate independently of the SODAS compiler. In the future this redundancy will be eliminated.

---

From the collection of the Computer History Museum (www.computerhistory.org)
SODAS Description

subsystem counter

sensor [1] ← sensor [1] ∧
reset ← reset ∧ light ← light [3] ∧
lt1 → L[1] ∧ leftturn → leftturn [1] ∧
cars → cars [1];

subsystem counter

reset ← reset ∧ light ← light [6] ∧
cars → cars [2];

subsystem counter

reset ← reset ∧ light ← light [9] ∧
cars → cars [3];

subsystem counter

sensor [1] ← sensor [16] ∧
sensor [3] ← sensor [18] ∧
reset ← reset ∧ light ← light [12] ∧
cars → cars [4];

Comments

{ counter for leg 1 - reference to component defined above

{ specification of connections to interconnectors

counter for leg 2

counter for leg 3

counter for leg 4
SODAS Description

subsystem

input Boolean array L[1:4];
output Boolean ltl


ltl ← L

BOOLE begin
input L[1:4];
element ltl;
gate ltl type or inputs L[1], L[2],
L[3], L[4];
outputs ltl;
end BOOLE description of joiner;

subsystem

input Boolean array outside light [1:12];
input integer array cars [1:4];
input Boolean loc, coc, reset, ltl;
input integer tmax, tmin, ttot, initialmode;
output Boolean array light [1:12]


loc ← loc ∧ co; coc ← reset ∧ reset ∧
light ← reset; light ← loc; light ← coc;
light ← outside light; light ← ltl;
light ← tmax; light ← tmin; light ← cycle time;
light ← initialmode; light ← cars

Comments

-- joiner

declaration of:
- inputs
- outputs
- connections
- immediate dependencies

BOOLE description of "or" gate

-- control unit

input declarations

output declaration

connections
to inter-
connectors

declaration
of immediate
dependencies
SODAS Description

SFD begin

comment control unit subsystem;

input Boolean array outsidelight [1:12];
input Boolean loc, coc, reset, ltl;
input integer array cars [1:4];
input integer tmax, tmin, ttot, initialmode;
output Boolean array light [1:12];
integer mode, tim, ctmax, ctmin, cttot;
real factor;

integer procedure newmode;
begin
integer i, high, nscars, ewcars, tcars;
nscars=cars[1]+cars[3];
ewcars=cars[2]+cars[4];
tcars=ewcars+nscars;
if tcars=0 then tcars=1;
if not ltl then
begin
high=0;
for i=1,2,3,4 do
if i = mode \ (cars[i]>0 V loc) then
begin
if cars[i]=high then
begin
high=cars[i]; newmode=i;
end;
end;
factor=high/tcars;
end else
begin
if mode = 5 V mode = 7 then
begin newmode=7; factor=5*nscars/tcars; end else
if mode = 5 V mode = 7 \ (leftturn[1]Vleftturn[3]) then
begin newmode=5; factor=5*ewcars/tcars; end else
if mode = 8 \ (nscars=ewcars) then
begin newmode=8; factor=nscars/tcars; end else
begin newmode=6; factor=ewcars/tcars; end;
end;
end of new mode procedure;
end;

integer procedure cycletime;
if mode = 5 V mode = 7 then cycletime = tmin
else if not loc then cycletime = tmax
else cycletime = factor*cttot;

Comments

control unit

now has more inputs and fewer internal variables

selects new mode to maximize the number of cars allowed to move

determines the time to be spent in the new mode
SODAS Description

procedure initializecontroller;
begin
integer i;
for i:=1 step 1 until 12 do light[i] := false;
mode := initialmode;
tim := timax; factor := 1;
ctmax := ctmax; ctmin := ctmin;
cttot := if loc then ttot else 1800;
ltl := false;
end initialization of the controller;

procedure setlights (a,b); Boolean array a; integer b;
begin
integer i;
for i:=1 step 1 until 12 do a[i] := false;
end setlights;

continue:
time begin
if reset then initializecontroller;
if coc then
begin integer i;
for i:=1 step 1 until 12 do
light[i] := outsideight[i];
go to done;
end;
if loc then cttot := ttot else cttot := 1800;
tim := tim + 1;
if tim>ctmax then
begin
tim := 1;
mode := newmode;
ctmax := cyctime;
setlights (light, mode);
end;
done: go to continue;
end of time block;
end SFD ALGOL description of control unit

Comments
initializes internal variables and outputs

sets traffic lights, given desired mode

this is the same control algorithm discussed in Figure , except cars is now an input and not an internal variable

end SODAS description of traffic control module
LEFTTURN [1] \rightarrow \text{SENSOR} [1:5]
CARS [1] \rightarrow \text{LIGHT} [3] \rightarrow L [1]
RESET

LEG 1 COUNTER AT SECOND LEVEL OF DESCRIPTION

---

Figure 8—Leg 1 counter at third level of description
the reader should note the strong correspondence between block diagrams and the structure of the SODAS program.

We consider this correspondence one of the most important features of SODAS and its sub-language BOOLE. There is a simple (mechanical) process to go from a block diagram to a SODAS description of a system, or from a logic diagram to a BOOLE description of a network. SODAS does not restrict or influence the designer's way of thinking—except possibly to encourage him to be more precise in his definitions.

CONCLUSIONS

We have presented a methodology for systems design, and a language that we feel is a significant aid to systems designers. The aid that SODAS provides is probably best indicated by the phrase Structure Oriented Description And Simulation. SODAS is unique in its ability to describe systems as structures of components that operate in parallel. It is also unique in its ability to simulate such parallel structures correctly. We feel that these two features are almost indispensable in a software system to aid in the systems design process. As evidence we offer the example of the traffic control module; although the system is a small and simple one by today's standards, SODAS appears to be a substantial aid to the design and description of the example. Interested readers may wish to compare it with the description of student solutions to the same problem.

As an example of one of the incidental benefits of such a system, consider the problem of grading students (or evaluating the work of members of a professional design team). In either situation it is extremely difficult to evaluate the work of the individual team members when part of the system fails or is not completed. It is extremely difficult to determine just who is at fault, and, more important, to evaluate the work of the remaining team members. With SODAS, the specification of part of a system can be substituted for the actual design in determining if the remainder of the system functions correctly. Further, since there are precise and testable specifications, it is possible to determine which components do not meet those specifications. It appears to us that this will be a valuable tool for the managers of systems design projects.

Our work on the example has indicated that the use of SODAS forces a discipline upon the system designer that could greatly improve the quality of his work and reduce the time needed to complete a project. We have found that this discipline carries over to design problems in which, for one reason or another, SODAS is not used (e.g., the design of the SODAS system itself).

REFERENCES

1 D L PARNAS
Sequential equivalents of parallel processes
Center for the Study of Information Processing Carnegie Institute of Technology Pittsburgh Pennsylvania

2 N WIRTH H WEBER
Euler: A generalization of ALGOL and its formal definition
Part I CACM 9 pp 13-23 January 1966

3 Ibid
CS 20 Technical Report Computer Science Department Stanford University Stanford California
Figure 10—Decoder
Figure 11—Gaiting for special counter
Figure 12—Memory for special counter
### SODAS Description

#### subsystem

**input** Boolean sensor, light, reset;
**output** Boolean ltl, leftturn, modsensor

reset ← reset ∧ ltl → ltl ∧
leftturn → leftturn [1] ∧
modsensor → modsensor

ltl ← reset;
leftturn ← reset;
ltl ← sensor; leftturn ← sensor;
modsensor ← sensor;
modsensor ← reset

**BOOLE begin**

*input* sensor, light, reset;
element not reset, ga, gb, jk, jl, ta, tb,

L[1:3], ltl, leftturn, notl, modsensor;

gate not reset *type not inputs* resets;
gate ga *type and inputs* not reset, light;
gate ga *type or inputs* sensor, leftturn;
gate mod sensor *type and inputs* ga, L3, not reset;
gate ta *type or inputs* sensor, Ll;
gate tb *type or inputs* jl, L3;
gate ltl *type and inputs* ta, not reset;
gate left turn *type and inputs* tb, not reset;
gate jl *type and inputs* sensor, not l;
gate not l *type not inputs* L3;
gate kl *type or inputs* mod sensor, reset;
flip-flop L1 *type jk inputs* sensor, reset;
flip-flop L2 *type jk inputs* jl, kl;
flip-flop L3 *type d inputs* gb;
*outputs* ltl, leftturn, mod sensor;
**end** BOOLE description of memory unit;

### Comments

- **memory unit**
  - declaration of inputs and outputs
  - declaration of interconnections
  - declaration of immediate dependencies

#### Note #1
- declarations of circuit elements and their interconnections

---

**Figure 13—Boole description memory unit**

From the collection of the Computer History Museum (www.computerhistory.org)
4 P NAUR (ed)  
Report on the algorithmic language ALGOL 60 (revised)  
CACM 6 pp 1-17 January 1963  
5 D E KNUTH J L McNELLY  
SOL—A symbolic language for general purpose systems simulation  
IEEE Transactions on Electronic Computers August 1964  
D E KNUTH J L McNELLY  
A formal definition of SOL  
IEEE Transactions on Electronic Computers August 1964  
6 D L PARNAS L C RICHARDSON W H KOHL  
Preliminary version—an introduction to Boole-66  
Unpublished manual available from Computation Center Carnegie Institute of Technology  
7 D L PARNAS  
A language for describing the functions of synchronous systems  
Comm ACM February 1966  
8 J FIERST (ed)  
ALGOL 2o language manual  
Computation Center Carnegie Institute of Technology Pittsburgh Pennsylvania  
9 J C STRAUSS D L PARNAS Y WALLACH R W SNELSIRE  
A design emphasis problem solving experience  
Department of Electrical Engineering Carnegie Institute of Technology  
10 IBM Corporation White Plains New York Introduction to General Purpose Systems Simulator III  
11 H M MARKOWITZ B HAUSNER H W KARR  
SIMSCRIPT a simulation programming language  
Prentice Hall Inc Englewood Cliffs New Jersey 1963  
12 O J DAHL K NYGAARD  
SIMULA—An ALGOL based simulation language  
Comm ACM pp 670-678 September 1966